UNCLASSIFIED

AD-E501 263
Copy 13 of 23 copies

DTIC FILE COPY (2)

IDA MEMORANDUM REPORT M-389

THE EUROPEAN FORMAL DEFINITION OF Ada - A U.S. PERSPECTIVE

AD-A225 519

Richard A. Platek

January 1988

DTIC
ELECTE
AUG23 1990
S D
E

Prepared for
Ada Joint Program Office (AJPO)

BEST
AVAILABLE COPY

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311

UNCLASSIFIED

90 08 22 075

IDA Log No. HQ 87-32838

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Group Reports

Group Reports record the findings and results of IDA established working groups and panels composed of senior individuals addressing major issues which otherwise would be the subject of an IDA Report. IDA Group Reports are reviewed by the senior individuals responsible for the project and others as selected by IDA to ensure their high quality and relevance to the problems studied, and are released by the President of IDA.

Papers

Papers, also authoritative and carefully considered products of IDA, address studies that are narrower in scope than those covered in Reports. IDA Papers are reviewed to ensure that they meet the high standards expected of refereed papers in professional journals or formal Agency reports.

Memorandum Reports

IDA Memorandum Reports are used for the convenience of the sponsors or the analysts (a) to record substantive work done in quick reaction studies, (b) to record the proceedings of conferences and meetings, (c) to make available preliminary and tentative results of analyses, (d) to record data developed in the course of an investigation, or (e) to forward information that is essentially unanalyzed and unevaluated. The review of IDA Memorandum Reports is suited to their content and intended use.

The work reported in this document was conducted under contract MDA 903 84 C 0031 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

This Memorandum Report is published in order to make available the material it contains for the use and convenience of interested parties. The material has not necessarily been completely evaluated and analyzed, nor subjected to formal IDA review.

© The Government of the United States is granted an unlimited license to reproduce this document.

Approved for public release; unlimited distribution.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	January 1988	Final

4. TITLE AND SUBTITLE
The European Formal Definition of Ada---A U.S. Perspective

5. FUNDING NUMBERS
MDA 903 84 C 0031
T-D5-304

6. AUTHOR(S)
Richard A. Platek

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Defense Analyses (IDA)
1801 N. Beauregard Street
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER
IDA Memorandum Report M-389

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office (AJPO)
Room 3D139, The Pentagon
Washington, D.C. 20301

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, unlimited distribution.

12b. DISTRIBUTION CODE
2A

13. ABSTRACT (Maximum 200 words)

IDA Memorandum Report M-389, *The European Formal Definition of Ada---A U.S. Perspective*, documents an evaluation of the European Economic Community's formal definition of Ada. The report includes an estimation of the benefits of U.S. participation in the Advisory Group which reviewed the formal definition work, describes what could be accomplished with the formal definition by interested U.S. parties, and presents an opinion of what should be done.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Ada Programming Language; European Economic Community (EEC); Ada Formal Definition; Semantics; Meta Language; SMoLCS.	56
	16 PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

UNCLASSIFIED

IDA MEMORANDUM REPORT M-389

THE EUROPEAN FORMAL DEFINITION OF Ada - A U.S. PERSPECTIVE

Richard A. Platek

January 1988

A-1

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 84 C 0031
Task T-D5-304

UNCLASSIFIED

Contents

Preface

The purpose of this report is to provide a basis for an evaluation of the EEC's Formal Definition of Ada. In particular, we are concerned with what role the U.S. can and should play in support of the European effort itself or, more broadly, in support of work which modifies, extends, or replaces the European accomplishment.

Chapters 1 and 4 provide our perspective on the Formal Definition and Chapters 2 and 3 serve as technical background to this perspective. In particular, in Chapter 4, we include our estimation of the benefits of U.S. participation in the Advisory Group which reviewed the FD work. We describe what could be done with the FD by interested U.S. parties and also present our opinions of what should be done.

I would like to personally thank all the members of the European Ada Formal Definition team and the other European members of the Advisory Group for their kindness and hospitality. Special thanks goes to Egidio Astesiano who has spent numerous hours with me explaining the fine details of his approach to concurrency. Prof. Astesiano was kind enough to invite me to visit Genoa to meet with his coworkers on this effort.

Chapter 1

Introduction

The purpose of this report is to provide a basis for an evaluation of the European Economic Community (EEC[1])'s Formal Definition (FD) of Ada. Its intended readership consists of government personnel and their outside supporting consultants who are tasked with managing the transition to Ada. This audience determines the report's scope and level of technical detail. In particular, we are concerned with what role the U.S. can and should play in support of the European effort itself or, more broadly, in support of work which modifies, extends, or replaces the European accomplishment.

When considering a U.S. response to the European FD, technical and policy concerns can not be cleanly separated. This is true of many Ada issues since Ada is not only a programming language, but also the centerpiece of a major, U.S. government-sponsored, software engineering initiative. Ada is mandated in certain classes of government (both defense and non-defense) software acquisitions. It is expected that such mandates will be extended to other classes; that waivers will be denied; and that Ada will, in the near term, become the principle acceptable computer language for U.S. government projects. It is further expected that the non-government, commercial community will independently adopt Ada once the supporting technology, training and culture is in place.

These language requirement policies are an attempt to reach a software standardization goal. The rationale supporting this goal, which we review in the next Section, is sound. Furthermore, it remains sound when restated within even larger, expanding contexts of U.S. economic and military alliances. If anything, standardization gains even more cogency when viewed in terms of these less integrated, more diverse, communities.

As an example of our point, consider NATO's oft-stated goal of system interoperability. Failure of interoperability is politically embarassing (the B1-B bomber could not take off from the 1987 Paris Air Show due to an electrical incompatibility with the French ground power supply[2].) Such embarassments undermine the perception of unity and erodes the Alliance's raison d'être—deterrence. On the other hand, interoperability is difficult to achieve since it must overcome various historical realities. Present day military and industrial allies developed independent processes and techniques as commercial and military rivals. A frequently mentioned example of such historical divergence in the NATO arena is the use of incompatible national

[1] EEC is the customary Anglo-American designation; in Europe it's the Commission of the European Communities, CEC.

[2] "Aviation Weekly", June 22, 1987.

Identification Friend or Foe (IFF) aircraft systems. Software standardization, on the other hand, actually stands a chance of success. The adoption of Ada (and the Common Apse Interface Standard, CAIS) by NATO is, thus, a policy decision of considerable technical and politico-military merit. It serves to strengthen the bonds of Western common defense by introducing a linguistic unity.

For these reasons, we feel that respectful consideration should be given to major allied sponsored Ada initiatives such as the EEC Formal Definition and that a positive and appropriate U.S. response be made. The EEC was an early supporter of Ada; beginning its sponsorship of the Ada Europe Organization in 1980. The most positive response would be support for a joint European/U.S. effort to extend the existing work. Such an effort could exploit the different strengths of each computer culture. We return to the question of recommendations in the last Chapter.

1.1 Standardization—the Rationale

The Ada programming language was developed as part of DoD's attempt to impose some order on an anarchic situation. Unregulated response to increased software demand resulted in an abundance of programming languages and dialects. While special purpose languages such as FORTRAN (scientific programming) and COBOL (business data processing) have undergone a standardization process over the years, general purpose languages such as Pascal continue to sprout incompatible dialects (even though Pascal has been "standardized") as they are adapted to respond to diverse application requirements. Software engineers are well aware that the same higher order language program can produce different behavior, not only on different hardware implementations, but also when translated by different compilers targeted for the same machine. Systems programming, in particular, is highly idiosyncratic and systems programmers need to be continuously aware of the strategies used by the particular compiler they are working with. This impacts both program portability and, perhaps more importantly, programmer portability.

Such lack of standardization is, of course, a reflection of the immaturity of the discipline of software engineering. Indeed, supporters of the present state of affairs (and there are many) argue that premature standardization throttles creativity. While one must be sensitive to the validity of this claim, it is also true that certain areas of software are sufficiently mature so as to

tolerate a degree of standardization. Ada is an engineering compromise designed to enforce such a tolerable degree of standardization. Being a compromise, it doesn't completely solve the problem. Instead, it tries to rationalize the situation by locating system specific information in one place and minimizing the quantity of such information by allowing a spectrum of alternative machine behaviors to be legal. A "correct" Ada program must be correct under all allowable behaviors. For systems programming, Ada provides standard interfaces to the hardware.

1.2 The EEC Formal Definition

The European Economic Community recently (1984 - 1987) undertook an effort to provide a formal mathematical definition of the Ada programming language which is informally defined by the ANSI (American National Standards Institute, Inc.)/DoD Standard, ANSI/MIL-STD-1815A 1983. Ada is also presently undergoing ISO (International Standards Organization) standardization. In addition to the Ada Reference Manual (RM) [ANS83], the EEC FD team gave consideration to the Language Maintenance Committee under ISO/SC22/WG9 and the Language Maintenance Panel under the DoD (essentially the same persons) who issue interpretations of the RM in response to questions asked. Some of these "interpretations" may conclude things not derivable from, or even contrary to, the RM.

The prime contractor for the EEC FD effort was Dansk Datamatik Center (DDC) of Denmark. DDC was supported by Consorzio per la Ricera e le Applicazioni di Informatica (CRAI) of Italy, an organization which attempts to build up an information technology capability in the South of Italy, together with various consultants such as Ugo Montanari from the University of Pisa, Dienes Bjørner from the Technical University of Denmark, and a group from the University of Genoa led by Egidio Astesiano. In order to successfully carry out such a multi-national, geographically disperse undertaking, it was necessary to divide responsibility. The static semantics were worked out primarily by DDC while the dynamic semantics were primarily the responsibility of the Italians working under the direction of Prof. Astesiano. Indeed, Prof. Astesiano's forceful personality dominated the project giving it a unity which it might not have otherwise attained.

As part of its supervisory responsibility for this effort, the EEC selected Prof. Joe Stoy of Oxford University and Dr. Otthein Herzog of IBM Germany as reviewers. In addition, an outside Advisory Group was set up. The

purpose of this group was to periodically review the results and goals and to offer criticism and other guidance. The group met four times at EEC Headquarters in Brussels, Belgium:

1. 20-21 March 1986
2. 26-27 June 1986
3. 6-7 October 1986
4. 23-24 February 1987.

The Advisory Group consisted of E. K. Blum, Manfred Broy, Robert Dewar, Gerald Fisher, Kit Lester, David Luckham, Robert Maddock, Richard Platek, Knut Ripken, Jürgen Uhl, and Martin Wirsing. No member of the Advisory Group attended all four reviews.

1.3 Report Overview

In addition to this introductory chapter, this report contains three others. Chapter 2 is a tutorial on the background material which underlies the formal definition itself. Unfortunately, the formal *definition* makes frequent appeals to a large body of previous work in formal semantics which is known only to specialists in theoretical computer science. This seems to be done in order to render the FD more accessible to experts in formal semantics. The FD would greatly benefit from being recast on a stand-alone basis without these frequent references to "denotational semantics", "algebraic semantics", "observational equivalence", etc. To do so, however, would require an extensive rewriting. In lieu of this, we offer our brief tutorial together with bibliographical references in order to orient the general reader.

In Chapter 3, we present an overview of the FD itself as a guide to someone who wishes to penetrate it further. In addition, we mention various secondary technical papers and reports which are spin-offs of the effort and are helpful to the thorough reader. Neither in this chapter, nor in others, do we undertake the task of debugging the definition. Our overview follows an EEC deliverable "A Project Overview" which was completed in June 1987.

In Chapter 4, we offer our perspective based on our rather limited participation in this effort. We include our estimation of the benefits of this participation. We describe what could be done with the FD by interested U.S. parties and also present our opinions of what should be done.

Chapter 2

Formal Semantics—A Sketch

Prior to the present EEC effort to provide a formal definiton for full Ada, there were several partial attempts which we review in the next Section. These concentrated on the sequential aspects of the language. The rationale for this approach was that techniques for modeling sequential languages were well known and it was felt that concurrency could be subsequently added on to a sequential semantics by using extra clauses in the definition. At the very begining of the current effort, Prof. Astesiano argued that such an approach was doomed to failure. Ada is a concurrent language whose concurrency impacts every aspect of the language. This includes not just patently concurrent actions such as task rendezvous, but also what are usually thought of as "sequential" actions such as declaration, evaluation, assignment, etc. The semantics of the latter must take into account ongoing parallel processes which might interfere with the so-called "sequential" actions. The result of assigning a value to a variable, for example, depends on the activity of other tasks which can access the variable.

In order to provide a mathematical framework within which a non-toy concurrent language can be formally modeled, a good deal of preliminary original mathematics was undertaken by Prof. Astesiano. For continuity sake, Prof. Astesiano felt it was necessary to relate his framework to contemporary, theoretical, computer science investigations into the modeling of concurrency, research into formal specification languages, etc. Unfortunately, the references in the FD to the theoretical computer science literature are not for quoting results which are then used. Rather, they serve the purpose of suggesting to the reader a rationale for what is going on; they elicit from the reader certain expectations of form and substance based on the reader's past experience so that the technical material need not be supplemented by extensive discursive, informal sections. This is a standard technique in computer science; Ada, for example, uses the well-known symbol ":=" for the assignment operation even though, as we have seen above, in the presence of concurrency its meaning is greatly changed. The reason for not introducing a new symbol even when a new operation is being introduced is to call into play all the reader or programmer knows about assignment. Unfortunately, the U.S. reader does not have the extensive grounding in theoretical computer science (his training is more in systems implementation) so that the "metaphorical" references to past developments in the literature may be more confusing than useful.

In this Chapter, we provide the basic background which will enable the reader to undertand the frequent references to the theoretical computer science literature scattered throughout the Formal Definition.

2.1 Why Formal Semantics?

Formulation of a Formal Semantics for Ada can be seen as an attempt to further codify the standardization effort beyond ANSI and ISO levels. The aim is to provide an unambiguous definition of the language. A formal definition of a language should make it theoretically possible to decide which tests have deterministic results and, in cases where the answer is affirmative, to predict the results of such tests. In addition, a useable formal definition must be such that such questions have feasibly computed answers.

At least three previous attempts to provide such a formal semantics for Ada were undertaken. An effort at INRIA in France attempted to write a Denotational Semantics for Preliminary Ada. The results of this effort are not very well documented (although a description is given in [Jon80]); a tool to navigate through this definition was to be designed at ISI.

In [Bjo80], a series of reports are given of a Danish attempt, also for Preliminary Ada, using the formal specification language VDM. This definition was actually used in building the DDC family of validated Ada compilers. This was not an automated use, that is, the formal definition was not mechanically transformed into a compiler. Rather, the formal definition was used by the compiler writers as the reference instead of a natural language requirements document. DDC personnel with whom we have spoken claim that the extra up-front effort taken to first formalize the definition of Ada led to efficiences in the long run. In particular, because abstraction played such a central role, it led to a compiler that was simple to re-target. The current EEC Ada FD effort was originally conceived as an extension of the original DDC work from Preliminary Ada to ANSI Ada, but this had to be abandoned in the face of the previously mentioned problems raised by concurrency. Nevertheless, the current EEC Ada FD tries to maintain some family resemblance to the original DDC work; the meta-language in which the definition is present is modelled after VDM.

Finally, the NYU project which produced the first validated Ada compiler, also had a formal definition goal. The very high level programming language SETL [SDS86] is very similar to a formal specification language; it is based on set theory. Thus, a formalized definition of Ada in SETL could serve both as a compiler and as a formal definition. Indeed, members of the SETL compiler team such as Ed Schonberg and Robert Dewar who were exposed to sections of the EEC Ada FD remarked on how similar parts were to the SETL compiler. Unfortunately, the compiler and formal definition goals of the NYU project diverged with the need for a validated

compiler dominating. Recently, an NYU graduate student, Brian Siritsky, has returned to the original goal of providing a formal definition of Ada in SETL.

Why does one need a formal semantics? Without formalism, it is impossible to say certain things precisely. The natural language RM, like any complex natural language statement (consider the text of legislation), is full of ambiguities, inconsistencies, and other semantically grey areas. In the absence of a precise mathematical framework for determining various meanings, debates among Ada experts tend to take on a legal form. (Indeed, such experts are known as "Ada lawyers".) Like a talmudic discussion, uncertainty about an issue is decided by a debate where arguments are presented in terms of the law (i.e., the RM) and various accepted interpretations of the law (i.e., the findings of the Language Maintenance Committee). In the debate, different Ada lawyers can make strong cases for opposite findings. A formal definition would turn such a discursive legal wrangling into a mathematical issue.

The questions at issue are not academic. Consider allowable optimizations. In Ada, the reordering and deletion of certain actions is allowed if there is a "guarantee that the effect of the program is not changed by the reordering[1]." A formal semantics for the language is needed in order to rigorously decide whether a proposed reordering is allowable. Because of the presence of exception raising, it is far from a simple matter to determine whether a proposed reordering changes the effect of a program. Consider the following rule governing allowable optimizations[2]:

> A predefined operation need not be invoked at all, if its only possible effect is to propagate a predefined exception. Similarly, a predefined operation need not be invoked if the removal of subsequent operations by the above rule renders this invocation ineffective.

Because of the lack of a precise, succinct formal semantics, it becomes very difficult to decide whether a particular optimization satisfies this rule.

The issue of optimization is just now emerging in the compiler community and various issues are being decided in an ad hoc manner.

In addition to these issues of compiler correctness, a formal definition could form the basis of a whole new generation of semantically based software tools. Today's software tools are largely syntactically based; they rely

[1] Section 11.6, para. 2 of the RM.
[2] Section 11.6, para. 7.

on the syntactical form of the language rather than on the meaning of the language's expressions and statements. A good example is a syntax directed editor which frees the user from having to memorize Ada's concrete syntax. The user can construct his programs directly in terms of the language's abstract syntax which is more regular, and hence easier to remember, than the concrete syntax. Such an editor provides both a concrete output for human consumption and a pre-parsed output for machine consumption. Some people find such editors a nuisance since they are so stupid (the editors not the people). To say they are stupid means that the editor has no knowledge of what it is doing other than syntactic knowledge. The more semantic knowledge these editors could access the more "intelligent" they would appear to users and, hence, the more acceptable. The editors produced by the Cornell Synthesizer Generator [RT84], which is based on attribute grammars, are a good example of these remarks. Attribute grammars were originally introduced as a means of providing semantics for the syntactic grammars known as context-free grammars. Tools based on attribute grammars are examples of semi-semantically based tools.

Other possible semantically based tools are program verifiers, run-time monitors, interactive compilers supporting user-directed optimization (Ada's **pragma** facility is a very rudimentary form of interactive compilation), etc. The reason for the "etc." is that no one knows what other kinds of tools would emerge if existing programming languages had formal definitions.

2.2 What is Formal Semantics?

The syntax of a programming language consists of the rules determining which sequences of characters constitute legal programs. Semantics consist of what the program means or does. Within the syntactical realm, certain aspects are sometimes called "semantics." To distinguish these terms, syntactical semantics is called **static** semantics, while the meaning of a program is called its **dynamic** semantics. Static semantics is necessary when the syntactical definition of legality requires access to the meanings of subtexts and contextual considerations. Some examples will make this clear. In strongly typed languages, subprograms are declared with signatures which declare the types of arguments. The syntactical task of determining whether a subprogram call obeys the signature declaration is a part of static semantics. If overloading is allowed, as in Ada, so that subprograms with different signatures can have the same name, then the resolution as to which call is

actually being made is again part of static semantics. In Ada, certain syntactical forms are also overloaded; for example, the expression "A(i)" might mean a call of the function A with argument i, a reference to the array A at index i, or the conversion of an object i to the type A; it is a task of static semantics to determine which. Briefly put, static semantics answers those questions that can be resolved at compile time (including the evaluation of static expressions such as "3*5") while dynamic semantics addresses those questions that can only be determined at run time (such as the evaluation of expressions such as "3*X", where X is a program variable).

Although an actual compiler might resolve the static semantics directly in terms of given target machine, it is good software practice to design a compiler as a series of translators the first few of which (lexical analyzer, context free parser, semantical analyzer) constitute what could be called a static semantics for the program. Design of such compiler "front-ends" is an area of computer science which has incorporated a good deal of theory and current practice is usually sufficiently abstract so that one can speak of the output of this analysis as a static semantics. In summary, the data structures and algorithms involved in static semantics are sufficiently well understood that a formal static semantics can be given in software (rather than in abstract mathematics) and indeed this is common practice in modern compiler writing.

The situation with dynamic semantics is less evolved. Continuing with our metaphor of a compiler, one can imagine that the code generation translator has been factored into two parts to support re-targetability. The first maps the resolved static semantics version of the program into code for a generalized intermediate machine and the second maps this code into code for the target hardware. If full re-targetability is really a goal, then the intermediate machine should be made as abstract as possible to cover a wide spectrum of actual machines. If the intermediate machine were sufficiently abstract, we could say the resulting code represents the dynamic semantics for the program. Using the language of the next section, this would be an example of an operational semantics definition.

Unlike the case of static semantics with its well worked out theory of LALR parsers and attribute grammars, there is no agreement as to what such an intermediate machine should look like. The world of actual hardware architectures is evolving so fast that any fixed choice of style for the intermediate machine would not be sufficiently abstract so as to cover all implementations.

The theoretician's solution to this dilemma is to use mathematics as

the "intermediate machine". A formal dynamic semantics maps the fully resolved program into a mathematical object. The latter is the meaning of the program. For example, a program that computes the factorial of its input is mapped to the mathematical factorial function which constitutes the meaning of the program. It is what the programmer had in mind when he wrote the program text. Of course, the program (as contrasted with the mathematical function) really doesn't work when the argument is too big. This can be taken into account by mapping the program to a partial function which is the restriction of the mathematical factorial function to a subset of its domain. In truth, all such meaning functions should be finite, but it is usual in mathematics to replace such finite objects by their infinite extensions in order to simplify the analysis.

The case of factorial is too simple to illustrate the method since the programmer intended his program to be the factorial. Finding the mathematical meaning of an input/output transducer, for example, requires inventing new mathematics to serve as an appropriate basis for formal semantics. (The mathematical objects are functions working on streams.) This creative activity is a non-trivial pursuit and a good deal of academic computer science research has focused on this problem. Before streams were invented, for example, there was nothing in mathematics of a similar nature.

The drawback to the "denotational" approach, mentioned above, is that most proposed denotational semantics are not sufficiently computational and cannot be efficiently automated. The result is that a full-fledged denotational presentation might not enable one to answer the kinds of questions expected of a semantics. An appropriate criteria to use when evaluating a proposed method of formal semantics is whether it can support the discovery of answers to questions in a reasonable time. The questions we have in mind are those questions (for example, compiler correctness or allowable optimizations) which prompted the exercise in formality in the first place.

2.3 Species of Formal Semantics

Formal semantics for programming languages exist in three species with various mixtures:

- Operational semantics
- Denotational semantics
- Algebraic semantics.

We have briefly alluded to the first two in previous sections.

2.3.1 Operational Semantics

The operational semantics of a program is given by describing a run of the program (or a family of runs in order to cover cases where the language does not determine various alternative steps such as the order of evaluation of expressions.) The runs are described at the level of the language in contrast to machine level descriptions. The RM provides an informal operational semantics for Ada. While the operational approach to semantics, either formal or informal, appears to be quite natural and intuitive, there are several drawbacks.

We first consider compositionality. Compositionality, or Frege's Principle as it is called in logic, is the criteria that the meaning of a composite be a function of the meaning of its components. In the programming language context, it means that the meaning of a program can be given as a function of its statements and the meaning of compound statements as a function of the simple statements and similarly with expressions. The RM is laid out according to this principle; first the meaning (elaboration) of declarations, then the meaning (evaluation) of expressions, then the meaning (execution) of statements and so on. The problem is, when trying to understand the meaning of a specific item, one finds oneself flipping back and forth through the RM. Why? The reason is that the meaning of the item in question is not just a function of its components, but also of another structure which in formal semantics is called the environment. The environment is a complex data structure which includes the declarations in effect, the visibility of such declarations, the concurrent tasks, and indeed all the information necessary to carry out the elaboration, evaluation or execution. Because the environment is only implicit in the RM and is not given an explicit status, one finds oneself going back and forth. So, the demand for compositionality requires the use of environments to supply the information needed to build up the meaning of a composite from its components. This fact is also true for the Algebraic and Denotational approaches. In the Operational approach, however, the environment tends to be more complex because the level of abstraction is lower. Put another way, since in the Algebraic and Denotational approaches the meaning of an item is at a higher level of abstraction than a machine state transition some of what would be included in the Operational environment is included in the Denotational or Algebraic meaning of the item. Hence, the Denotational or Algebraic environment

is smaller and less complicated. In summary, in the other two approaches there is more of a balance of complexity between the meaning of the item and the environment.

A second drawback of the Operational approach is its specificity. The abstract machine being used is not abstract enough (this was mentioned before.) The generally accepted solution to this lack of abstraction is to also define a class of observation functions on the abstract machine. Then the meaning of a program is an equivalence class of runs on the machine where two runs are equivalent when they look the same with respect to the observation functions. This notion is known as **observational equivalence**. One can argue that the description in the RM is really of this form; there is an underlying implicit abstract Ada machine and an underlying implicit class of observational functions. This approach is the method of choice when describing concurrent systems; most of the Denotational and Algebraic approaches to concurrency have been found not to be as flexible as using observational equivalent runs of a model of a machine. This is not too surprising; concurrency is a very "operational" idea. The new mathematics necessary to really deal with concurrency is still in the process of being invented.

2.3.2 Denotational Semantics

A standard reference for Denotational Semantics is [Sto77], its author, Prof. Joe Stoy, was the Chairperson of the Advisory Group for the EEC Ada FD. In Denotational Semantics, mathematical objects such as sets, relations, functions, streams, are assigned to programs in a compositional manner. The meaning functions map program text into spaces of these mathematical objects. The reason the term "spaces" is used for the targets of the meaning functions is that these target sets usually come equipped with a topology in the mathematical sense. Topologies enable one to speak about **continuous** functions between sets and indeed continuity plays a large role in Denotational Semantics.

The topologies also allow one to talk about **limits** in sets. Such limits are important when defining the meanings of recursive functions and recursive data types. An example of such a type in Ada is an access type whose designated type is a record one of whose fields is of the given access type. Such records are introduced using an incomplete type declaration. An example is:

type CELL;

```
type LINK is access CELL;
type CELL is
     record
          VALUE : INTEGER;
          SUCC  :  LINK;
     end record;
```

It is easy to find a value for a variable of such a type which points to a record whose corresponding access type field has the same value. Such a value is a pointer to a circular structure. Continuing with our example the following program text creates assigns a value to HEAD which accesses a record whose VALUE field has value 0 and whose SUCC field has value equal to the value of HEAD. HEAD points to an infinite linked list of 0's.

```
HEAD : LINK := new CELL;
begin
     HEAD.VALUE := 0;
     HEAD.SUCC  :=  HEAD;
```

In Denotational Semantics, the denotation of the object accessed by HEAD would be an infinite mathematical object found as the limit of a sequence of partial approximations. This illustrates the point made earlier that in Denotational Semantics the meanings of items are generally more complicated than the meanings found in Operational Semantics. In Operational Semantics, the meaning would be the finite data structure described above while in Denotational Semantics it is an infinite mathematical object.

While Denotational Semantics is quite elegant from the mathematical point of view, it is difficult to automate. The target spaces include not just the limits which come up in computation, but all limits. Thus, the space [[CELL]] of values for the type CELL in the previous example is quite large while the function space of continuous maps from [[CELL]] to [[CELL]] which is used to provide a semantics for Ada **functions** from CELL to CELL is even more complicated.

2.3.3 Algebraic Semantics

In contrast to Denotational Semantics' use of the full machinery of mathematics, Algebraic Semantics restricts itself to a much smaller portion. As mentioned above, Denotational Semantics uses the notions of limits and continuity; Algebraic Semantics uses only algebra which has more finitude.

The kind of algebra used is based on axiomatic equational logic. A textbook introduction to the subject is presented in [EM85]. The Affirm verification system uses an equational logic specification language.

In the Denotational approach, one assumes and uses the properties of sets, relations, functions, topologies, and continuity. In the Algebraic approach, one assumes much less. Everything needed is introduced axiomatically and all axioms take the form of equations. The equations are between terms built up from variables, constants, and function symbols. The variables in question are "logical variables" as distinguished from programming variables which correspond to memory locations. Logical variables are like those used in mathematics to make general statements as in the equation $x + y = y + x$. The variables range over declared *sorts*; the sorts are unspecified types, that is, we do not assume any meaning for them. They are analogous to Ada incomplete or private types. Some of the sorts are actually parameters for the specification. They are analogous to Ada generic private types. Each constant is also of fixed sort and each function has a fixed signature. We illustrate with an example.

Suppose we wished to axiomatize the notion of finite lists over a set. Since we would like our specification to be abstract with respect to this underlying set, we treat it as a generic sort *elem*. Instead of saying what the lists are, we assume another sort *list* together with a constants *nil* of sort *list*; a function *cons* which tacks an element onto the front of a list; a function *append* which appends two lists; and similarly if we wished a function *reverse* which reverses a list, etc. Each of these functions are presented by giving their signature and some equational axioms about them. The only knowledge we are supposed to have about these functions are the further equational consequences of the axioms. The fact that the functions are given conventional names (i.e., *nil*, *cons*, etc.) could be misleading; we are not allowed to use any properties of these functions except those which can be deduced from the axioms.

We now illustrate the above in a style similar to Ada. The following is a sample specification:

generic
    sort *elem*;
specification LISTS **is**
    sort *list*;
    constant *nil* : *list*;
    function *cons* : *elem*, *list* → *list*;

function *append* : *list*, *list* $\rightarrow$ *list*;

axioms

variable e : *elem*;

variable x, y : *list*;

equations

$$append(nil, x) = x; \tag{2.1}$$

$$append(cons(e, y), x) = cons(e, append(y, x)); \tag{2.2}$$

end axioms;

end specification LISTS.

We should emphasize that this is not a program for *append*; it's a specification. The only properties of the functions that we are allowed to use are those that follow logically using the properties of equality. The rules of equality logic include substituting terms for variables in equations and replacing a term in an equation by another term which has been proved equal to it. For example, we can deduce

$$append(cons\ (e_1, cons\ (e_2, x)), y) = cons\ (e_1, cons\ (e_2, append\ (x, y))) \tag{2.3}$$

for arbitrary e_1 and e_2 of sort *elem* and x, y, of sort *list*. This follows by the following deduction. First, simultaneously substitute e_1, $cons(e_2, x)$ and y, for e, y, and x respectively in equation (2) to get

$$append(cons(e_1, cons(e_2, x)), y) = cons(e_1, append(cons(e_2, x), y)). \tag{2.4}$$

Now, again substitute into equation (2); this time substitute e_2, x, and y for e, y, and x respectively to get

$$append(cons(e_2, x), y) = cons(e_2, \text{append}(x, y). \tag{2.5}$$

The left hand side of equation (5), $append(cons(e_2, x), y)$ is a subterm of the right hand side of equation (4). Replacing the latter with the right hand side of equation (5) yields the desired equation (3).

Much research in automatic thoerem proving during the last few years has focused on term rewriting methods for equational logic, [Les87]. The proof given above could be accomplished fully automatically.

In contrast, the equation

$$append(x, append(y, z)) = append(append(x, y), z)$$

is not provable from equations (1) and (2) just using equality logic. Its proof requires structural induction on x which is justified if we add to the specification the fact that the sort *list* is the initial algebra generated by *nil* and *cons*. All the necessary material on such initial algebras is presented in the previously mentioned text [EM85]. The example illustrates the main drawback of the Algebraic approach. If we can't prove an equation like the last, then one has to go back to the specification and add it as a new equational axiom. Such additions occur frequently as one discovers that one's axiomatization is too weak. Because of this frequent modification of the starting point, errors have a way of sneaking in. By error, we mean the addition of an equation which is not true for the intended meaning. Adding such equations does not make the specification inconsistent (no equational logic axiomatization is inconsistent, there is always the one element model!), so that it might not become apparent that one has wandered into error. The axioms are not inconsistent, but they might not hold for the notion one is trying to axiomatize.

Using the Algebraic approach, one can indeed build up in a systematic manner all the notions one needs for a semantics. The Algebraic approach appears to be the method of choice among European workers in formal semantics. It's attraction is the simplicity of the underlying logic, viz. equations. It can be extended to equations using partial functions and to conditional equations which have the form

$$t_1 = t_2 \rightarrow t_3 = t_4$$

which means that the equation on the right is assumed to be true for those values of the variables which make the equation on the left true. Since the terms might contain partial functions, the real meaning of the conditional equation is: for all values of the variables which make both t_1 and t_2 defined and their values equal, it is the case that both t_3 and t_4 are defined and equal. The formula says nothing about any other cases. More generally, the hypothesis of a conditional equation (i.e., the left hand side of $\Rightarrow$) can be a finite set of equations. The meaning is, for all values of the variables if all the equations in the hypothesis set are true (meaning both sides of each equation is defined as having the same val) then the conclusion is true. These kind of conditional equations with partial functions is the style used in the Ada FD.

If, for some reason, one needed a Denotational style semantics, but was forced to work within an Algebraic framework, then equational logic could

be used to axiomatize all the fundamental Denotational notions. There are axiomatic equations for sets (in the style of our LISTS specification given above), functions, approximations, limits, etc. The foundations of the Denotational style need be given only once within Algebraic Semantics; from that point on, it is possible to proceed in Denotational Semantics. This is the style used in the Ada FD.

2.4 Concurrency

One of the most influential works in the semantics of concurrency is Robin Milner's Calculus of Communicating Systems, CCS, described in [Mil80]. A more recent reference is Milner's lectures [Mil85].

A CCS process is a possibly infinite sequence of events; events can be intraprocess computation events, interprocess synchronization and communication events, or system input/output events. Processes can also be nondeterministic. This is modeled by allowing a process to be all finite or infinite paths through a tree rather than a simple sequence. These paths are called the traces of the process. Trees begin at a start node labeled by the name of the process; arcs between nodes are labeled by event symbols chosen from an alphabet Σ; forks in the tree represent alternative ways of proceeding.

The process trees are generated by process descriptions. These descriptions are built up from simple processes by process constructors. For example, if P is a process and a is an event symbol in *Sigma* then $a.P$ is the process that begins with an a and then continues the way P does. If P_1 and P_2 are processes then $P_1 + P_2$ is a forking process; after a nondeterministic choice, it either proceeds like P_1 or like P_2. Again, if P_1 and P_2 are processes, then $P_1 \| P_2$ is the parallel interleaved combination of P_1 and P_2 with no synchronization or communication. The only atomic process in this algebra of process descriptions is **NULL** which does nothing. Thus, for example, $a.(b.\mathbf{NULL} + c.\mathbf{NULL})$ is the process which begins with the event a and then makes a choice to either undergo event b and then halt or to undergo event c and then halt. The traces of this process are a, b and a, c. In contrast, the process $a.(b.\mathbf{NULL} \| c.\mathbf{NULL})$ undergoes event a followed by an interleaving of b and c. Its traces are a, b, c and a, c, b.

Using the above process constructors, only finite processes can be described. To get infinite processes, recursion is required. Such recursively defined processes are defined using a *mu* operator. If E is a process description in which the process variable P occurs, then $\mu P(E)$ is the process

P which solves the recursion equation $P = E$. Some examples will clarify this. Consider $\mu P(a.P)$. This is the process P which solves the equation $P = a.P$ which begins with an a and then continues with P which in turn begins with an a and then continues with P, etc. Thus, P is the infinite process $a, a, a, \ldots$ which is an infinite sequence of a's. Its traces are all finite and infinite sequences of a's (there is only one infinite sequence). The process $\mu P(a.P + b.P)$ begins by non-deterministically chosing an a or a b and then starting over again. Its traces are all finite and infinite sequences of a's and b's.

Synchronization and communication are included by first introducing a silent even τ, pronounced "tick," and then including for each event a in Σ a complementary event $\overline{a}$. The resulting event set Λ consists of a and $\overline{a}$ for each a in Σ together with τ. Then, a new parallel combinator $P_1|P_2$ is defined with the property that in addition to interleaving, complementary events combine to form a τ. Thus, the traces in $a.\mathbf{NULL}|b.\overline{a}.\mathbf{NULL}$ which we will call Q for convenience are $a, b, \overline{a}$ and $b, a, \overline{a}$ and $b, \overline{a}, a$ and also b, τ.

If we now introduce a restriction operation $P\backslash A$ where A is a subset of Σ to mean we neglect transitions labeled either by events in A or their complements, then the only trace in $Q\backslash\{a\}$ is b, τ so that in this process synchronization between a and $\overline{a}$ is forced. This is a standard technique. Certain events like a are introduced just for synchronization purposes; one forms parallel combinations generating the silent synchronization events and then removes those synchronization signals which do not pair up correctly.

A further operation is relabeling, $P[S]$, where S is a map from Σ to itself. This changes all the labels a on the transitions to $S(a)$.

The semantics for these process descriptions are given in an Operational style. One generates all labeled transitions of the form

$$P_1 \xrightarrow{s} P_2$$

where P_1 and P_2 are process descriptions and s is an event symbol from Λ. The meaning of the labeled transistion is that the process P_1 undergoes event s and then continues as P_2 does. The labeled transitions are generated by starting from all transitions of the form

$$s.P \xrightarrow{s} P$$

and then generating new transitions using rules corresponding to the basic connectives. Some of the rules are:

- From
$$P_1 \xrightarrow{s} Q$$
conclude
$$P_1 + P_2 \xrightarrow{s} Q + P_2$$

- From
$$P_2 \xrightarrow{s} Q$$
conclude
$$P_1 + P_2 \xrightarrow{s} P_1 + Q$$

- From
$$P \xrightarrow{s} Q$$
and s not in A conclude
$$(P\backslash A) \xrightarrow{s} (Q\backslash A)$$

- From
$$P_1 \xrightarrow{s} Q_1$$
and
$$P_2 \xrightarrow{\bar{s}} Q_2$$
conclude
$$(P_1|P_2) \xrightarrow{\tau} (Q_1|Q_2).$$

The worked out examples in Milner's lectures and other places show how to model various concurrent situations using CCS. A related development is Hoare's CSP presented in [BHR84].

An interesting point is that the above operational semantics for CCS can be developed in Algebraic semantics using conditional equations with partial functions. This is the style used in the Ada FD.

One starts by introducing the sort *process* with constant **NULL** and functions over it corresponding to the process constructors. The constructor $a.P$ takes two arguments. The first is of sort *event*, the second of sort *process*. The constructor $P\backslash A$ also takes two arguments; the first of sort *process* the second of sort *set(event)*. The latter sort is formed using a generic SETS specification instantiated with *event*. To get labeled transitions, one first introduces a specification for Boolean logic based on a sort *bool*, constants

true and *false* and the well known truth functions *and*, *or*. etc. Then, one considers the labeled transition relation

$$P \xrightarrow{s} Q$$

as a partial function *trans* of signature

$$trans \quad : \quad process, event, process \rightarrow bool.$$

The intended meaning is that

$$P \xrightarrow{s} Q$$

is true in the operational semantics if and only if the term

$$trans(P, s, Q)$$

is defined and equal to *true*. This is accomplished by transforming each the rules, given above, for generating the operational semantics for labeled transitions into conditional equations. Some examples are

$$trans(s.P, s, P) = true$$

$$trans(P_1, s, Q) = true \Rightarrow trans(P_1 + P_2, s, Q) = true.$$

In this way, the translation of every true labeled transition will be a consequence in equational logic of the translations of the generating rules. In this way, CCS and its various descendants are codifiable in Algebraic Semantics.

Chapter 3

A Guide to the European Formal Definition

We begin our guide through the formal definition by reproducing the Forward provided with the FD. This Forward is reproduced at the start of each of the eight volumes constituting the FD. The Forward includes the project's stated goals against which it should be evaluated. It is also given verbatim since it serves to illustrate the criticism that the narrative portions of the FD, although written in English, are somewhat hard to follow by Americans.

> This project was initiated in 1984 with the aim of producing a formal definition of the language described in "Reference Manual for the Ada Programming Language ANSI/MIL-STD 1815A Ada" (RM) released January 1983.
>
> The project tries to fulfil the need for a description of the language without the deficiencies of a natural language description, like ambiguities and omissions. Generally the project also tries to solve the problem of getting an extensive and complex design as Ada onto paper in such a manner that computer professionals will be capable of reading and understanding the whole of the design, and at the same time use it as a reference document answering all questions about Ada unambiguously.
>
> The draft basis of this project, was to a large extent due to discussions in the Ada Europe working groups on Formal Semantics and Formal Methods. These Ada Europe working groups are one of the signs showing interest in Ada by the Commission of the European Communities.
>
> In order for the project to have some advise and criticism along the performance of the project, and international advisory group was formed. They are people from the international Ada scene, and the input from this group has been greatly influencing the work of the project.
>
> The main goals of the project may shortly be listed as follows:
>
> - Ease of use.
>
> The project tries to address a rather large audience, and hence the formal definition is not only formulas, but also a large amount of user information. Finally this information will include:
>
> - natural language description of the objective of each formula
> - natural language explanation of the function of each formula

- a description on how this formula relates to what parts of the RM
- a cross reference to what other formulae make use of this formula, and what other formulae are used by this formula

all of this information should allow a user with even minimal training in formal methods to access the formal definition.

- Mathematically correct.

  Another of the main goals is to make the formal definition mathematically correct. this ensures that this work will be useful for further research into the programming language Ada. One of the research areas, where this property is particularly necessary, is the area of mechanized use of the definition, as in automatic proof systems, and executable formal definitions.

  This correctness must also be extended to the fact, that all the permissiveness of the RM must be modelled faithfully, and the ambiguities be pointed out.

As a secondary benefit, the project has been a practical exercise in using a formal methodology on a large scale project. This is a very valuable experience, as we now see emerging technologies for designing, and constructing software in an industrial size using formal methodologies.

The definition itself may be seen as composed by several parts: the underlying methodology, the formal definition of Ada, the metalanguage used, and the toolset.

The formal definition of Ada is split into two parts: the static semantics and the dynamic semantics. Static semantics describes in detail all static properties (i.e., the checks and evaluations done at compile time) as described in the RM. Similarly, the dynamic semantics describes the effect of running a specific Ada program.

In order to have a metalanguage with the necessary properties, and power as well as being mathematically precise and correct, an algebraic definition language was chosen. This language is used to describe the static semantics of Ada, but it needs refinement in order to be able to model stronger concepts (e.g., the ability to model parallel processes) easily. This extension to

the metalanguage with the desired properties was defined using the SMoLCS (Structured Monitored Linear Concurrent Systems) methodology from the University of Genoa. Both parts of the metalanguage are used to model the dynamic semantics of Ada.

The metalanguage has been designed to be similar to programming languages, in order for computer professionals to have an easier access to the definition. Besides these consideration, an attempt has been made to keep the language as close as feasible to a known formal definition language: Meta-IV from the VDM methodology.

Finally a toolset has been designed, and implemented in order to handle the massive amount of information found in the formal definition of Ada. This toolset is providing syntax analysis, and type checking of formulae written in the metalanguage, as well as allowing for browsing through the definition.

3.1 Project Overview

A June 1987 deliverable "The Draft Formal Definition of Ada: A Project Overview", by J. Storbank Pederson of DDC provides a good summary of the structure of the Ada FD. We largely follow it in our presentation in this chapter quoting it extensively (with modifications) without explicit attribution. The references for the next few sections have been collected together in the "References" section. Citations to references in that section have the form [[Astesiano 87]].

The first part of the project was a test phase in which an underlying model of Ada was constructed. A methodology and a meta-language for the definition was then defined with the aim of getting a mathematically well-founded frame suitable for the definition of Ada. Finally, a trial formal definition of a subset of Ada was made in order to assure the expressive power of the proposed techniques.

The second part of the project was the full formal definition of Ada. Although this was the first attempt to give a precise formal definition of a language the size of Ada, it was facilitated by the knowledge obtained in the test phase. A preliminary study was made of the feasibility of using the AdaFD to prove certain aspects of the ACVC test suite. Moreover, it was demonstrated that the AdaFD can be executed. This was done using the RAP equational logic theorem prover from the University of Passau.

In parallel with the actual mathematical development of the AdaFD, it was described in English and correlated with the reference manual.

In brief, the main objective of the project was to produce a concise formal definition of the full Ada language. The formal definition can serve as a reference document for questions on Ada and as a basis for compiler development, compiler validation, and the writing of concise, informal descriptions of Ada.

Following the VDM-tradition (Vienna Development Method, [[Bjorner et al. 78]]) of programming language semantics, a formal definition consists of:

- An abstract syntax called AS1.
- A static semantics (well-formedness) definition based on AS1.
- An abstract syntax called AS2.
- A transformation from AS1 into AS2.
- A dynamic semantics definition based on AS2.

The abstract syntax, AS1, describes the input to the static semantics. In the AdaFD, it consists of a number of algebraic specifications. It reflects the structure of the BNF-grammar defining the concrete syntax of the language, but syntactic items, like keywords, that are present in the BNF-grammar to help in the parsing of the program text are not present in AS1. Similarly, information on operator precedence has been used to represent expressions in AS1 in a tree-like fashion.

The static semantics define the context sensitive conditions that a given AS1 construct must satisfy. These conditions can also be characterized as those expressable without reference to execution. The conditions may also be described as those Ada rules whose violation must be detected by a compiler. In language reference manuals, such conditions are expressed in natural language, typically using words such as "must", "allowed", "legal" or "illegal".

In the VDM-style, the static semantics is expressed by a set of, typically appicative, formulas in a denotational syntax-directed way, so that for each syntactic construct there is a formula expressing the well-formedness thereof using some kind of context information.

The abstract syntax, AS2, is chosen so as to be suited for expressing the dynamic semantics (run-time behavior) of a program. For simple languages,

AS2 may be the same as AS1. But, for complex languages like Ada, it would be very cumbersome to use AS1 as a basis for describing the dynamic semantics.

Introducing an abstract syntax different from AS1 requires the relation between AS1 and AS2 to be given. This is expressed in the form of a set of transformation formulas mapping AS1 constructs into AS2 constructs. These formulas, of course, utilize the formulas defined in the static semantics for resolving overloading, etc. They have not been formally defined within this project.

The dynamic semantics describe the run-time behavior of statically correct programs. In language reference manuals, this behaviour is expressed in natural language and the activities involved are described using terms as "evaluation" (of expresions), "elaboration" (of declarations) and "execution" (of statements). The existence of tasks in Ada has important impact on the dynamic semantics of Ada. Roughly speaking, the dynamic semantics can be divided into a pseudo-sequential part and tasking or parallel part, where the former resembles the dynamic semantics of a traditional sequential language like Pascal.

3.2 The Meta-language

It was foreseen, from the start, that the formal definition would contain a large number of formulas, especially in the parts related to the semantic information structures used in the static as well as the dynamic semantics. This called for a structuring mechanism and, since Meta-IV definitions are (formally) "flat", new features had to be introduced. The solution chosen was to define a meta-language for algebraic specifications. Moreover, modules were introduced consisting of a number of algebraic specifications, types that are particular algebras based on the algebraic specifications, and applicative formulas with parameters of those types. Traditionally the axiomatic specification techniques, to which algebraic approaches belong, are seen as "competitors" to the model-oriented approach, to which VDM belongs. But, it was felt that a useful result could be obtained by combining the two approaches, thereby allowing the user to use the algebraic parts when an axiomatic definition was appropriate, and to use the model-oriented parts in other cases. The way this is achieved is by first recognizing that the Meta-IV domain constructors are useful and essential facilities when constructing a model, and then defining parameterized algebraic specifications for all of

the domain constructors, so that when the parameters are supplied, a new algebraic specification is generated containing all the traditional operations on values of the sort of this new specification. Moreover, special syntactic constructs have been provided for expressing the semantics of tasking following the SMoLCS practical methodology. The meta-language is documented in [[Reggio et al. 86]].

3.2.1 Formatting Schemes

Formal definitions generally have a reputation for being hard to read and understand. In order to overcome part of that difficulty, a set of rules for writing the formal and informal parts of our definition were formulated. The rules, called formatting schemes, define for the different kinds of formulas present in the formal definition, how they must be structured and what kind of explanatory information must or can be provided. This ensures uniformity throughout the formal definition. As part of the formatting schemes, it was decided to present the informal explication of the formulas as an integral part of the formal definition, rather than in a separate document. This in our experience makes the formal definition more intelligible. The formatting schemes are documented in [[Astesiano et al. 86a]].

3.3 The Static Semantics Definition

Due to the concepts of separate compilation and program libraries, the formal definition of the Ada static semantics covers more than just the traditional context conditions. In a sense, the definition can, at the top level, be divided in two parts: one describing the effect of compiling a compilation, given a library; and one defining the well-formedness of a main program and all units referred to transitively by the main program, knowing that these units are either present in the library (they were compiled) or not present (they need to be compiled). The first part, which is the major part of the static semantics definition, describes whether and how the library changes as the result of compiling a compilation. This involves checking whether the context conditions are satisfied for the units being compiled in relation to the library.

Ada language constructs are represented in the static semantics definition by an abstract syntax called AS1. AS1 was defined in such a way that the AS1 representation of an Ada construct can be generated from the Ada text by a parser-like tool without any semantic analysis. This implies that

constructs that are potentially syntactically ambigious in Ada (like type conversions, function calls and indexed components) will not be disambiguated in AS1, but will be disambiguated during the semantic analysis. In general, AS1 is kept close to, and derived in a straightforward way from, the Ada syntax. The only real change made is that the operator precedences and the rules on parsing sequences of operators of the same precedence level, are used to transform "flat" expression structures into tree-structures.

The static semantics definition uses a number of information structures to hold the context information needed when expressing the well-formedness of the individual AS1 constructs. These structures are combined into a composite structure called "the surroundings" that includes components for expressing, for example, the scope and visibilty rules and the strong typing rules of Ada. These structures are defined using algebraic specifications. They define the operations available on the structures.

The static semantics is documented in [[Botta et al 87]].

3.4 The Dynamic Semantics Definition

The AS2 used by the dynamic semantics is close to AS1 and, hence, to Ada. This was the result of a trade-off between having an AS2 that made it easy to see the relation between the RM and the dynamic semantics, and having one that made the dynamic semantics formulas more elegant. AS2, however, has unique names and overloading resolved, and some unnecessary information has been removed, e.g. package renamings (for giving a new name to a package).

The approach taken in defining the dynamic semantics is described in [[Astesiano et al. 86]]. It is based on the principles of SMoLCS (Structured Monitored Linear Concurrent Systems) developed by Professor E. Astesiano at the University of Genoa.

Following the principles of [[Astesiano et al. 86b]] the dynamic semantics is divided into two steps:

- the first step, called the denotational clauses, associates terms, in a language suitable for representing processes, to AS2 constructs,
- the second step gives semantics to terms in the above language by an algebraic specification of a so-called "concurrent algebra" that represents a concurrent system.

A concurrent system is seen as a labeled transition system, that may be built hierarchically from a number of subsystems. The state of a system consists of the states of the subsystems and some global information. The global information contains a model of Ada storage and other information needed by more than one subsystem. The transitions of a system are described based on the transitions of its subsystems and some global information. The transitions of a system are described based on the transitions of its sub-systems in three steps:

Synchronization - defines the transitions representing synchronized actions of sets of subsystems and their effect on the global information.

Parallelism - defines the transitions representing the allowed parallel executions of sets of synchronized actions and the compound transformations of the global transformation (mutual exclusion problems, for example, are handled here).

Monitoring - defines the transitions of the overall system respecting some global constraints (e.g. interleaving, free parallelism, priorities, etc.).

The atomic actions are at the bottom of this hierarchy. They describe indivisible semantic effects of parts of Ada constructs. This means that when the set of all different atomic actions of the dynamic semantics were chosen they had to respect the granularity of the level of interruptabilty defined in the RM.

The information structures used in the dynamic semantics are defined using algebraic specifications. All of step 2 is also algebraically defined, but a special syntax has been used to express the axioms. Step 1 consists mainly of applicative formulas that in a syntax directed way follow AS2 and "provides input" to step 2. Since suggestive names have been chosen for the atomic actions and the operators of the concurrent algebra (part of the semantics associated to AS2 constructs by step 1), the first step can be read and intuitively understood by a reader, without the reader having to get troubled by the underlying semantics.

The dynamic semantics is documented in [[Astesiano et al. 87]].

3.5 References

[[Astesiano et al. 86a]] "The Draft Formal Definition of Ada, Formatting Schemes for the Final Definition", DDC/CRAI/University of Genoa, July

1986

[[Astesiano et al. 86b]] "The Draft Formal Definition of Ada, Generalities of the Underlying Model", CRAI/University of Genoa, December 1986 (part of [[Astesiano et al. 87]])

[[Astesiano et al. 87]] "The Draft Formal Definition of Ada, The Dynamic Semantics Definition" DDC/CRAI/IEI/University of Genoa, January 1987.

[[Bjorner et al. 78]] "The Vienna Development Method: The Metalanguage" Springer Verlag, Lecture Notes in Computer Science, Vol. 61, 1978.

[[Botta et al. 87]] "The Draft Formal Definition of Ada, The Static Semantics Definition", DDC, January 1987.

[[Gallo et al. 87]] "Ada FD Tool Set: Design", CRAI, January 1987

[[Reggio et al. 86]] "The Draft Formal Definition of Ada, The User Manual of the Metalanguage", CRAI/University of Genoa/IEI, December 1986.

3.6 Reports Produced by the Ada FD Project

The following are by Astesiano et al.:

"Static Semantics of 'Difficult' Example Ada Subset", March 1986

"An Extract from the Paper "Static Semantics of 'Difficult' Example Ada Subset"", March 1986

"Dynamic Semantics of 'Difficult' Example Ada Subset", February 1987

"An Extract from the Paper "Dynamic Semantics of 'Difficult' Example Ada Subset"", February 1987

"Extract from "The Static Semantics Definition" and "The Dynamic Semantics Definition"", May 1987

"Formatting Schemes for the Ada Formal Definition", July 1986

"The Draft Formal Definition of Ada, The Dynamic Semantics Definition", January 1987

"A User Manual of the Metalanguage for the Trial Definition", January 1986

"Formatting Schema for Semantic Clauses in the Trial Definition", September 1985

"The Ada Challenge for New Formal Semantic Techniques", November 1985

"Toward a SMoLCS Based Abstract Operational Model for Ada", August 1985

"A Syntax-Directed Approach to the Semantics of Concurrent Languages", October 1985

"Comparing Direct and Continuation Semantics Styles for Concurrent Languages", September 1986

"Related Structures and Observational Semantics - A Simple Foundation for Algebraic Specification of Concurrency", November 1985

"A Guided Tour To Ada FD (Dynamic Semantics)", April 1987

"Planning Ada Ada FD Courses, A Feasibility Study", April 1987

C. Bendix Nielsen, E.W. Karlsen: "Draft 2 Formal Definition of Ada Dynamic Sequential Semantics", September 1986

D. Bjorner et al. "The Role and Scope of the Formal Definition of Ada", March 1987

N. Botta et al. "A Note on Axiomatized Data Types from the Static Semantic", June 1986

"Bits and Pieces of the Formal Definition of Ada Static Semantics", September 1986

"The Draft Formal Definition of Ada, The Static Semantic Definition", January 1987

P. Christensen et al. "Dynamic Semantics Example Ada Subset", June 1985

A. Fantechi et al. "Feasibility of a ACVC Validation with respect to the Ada Formal Definition", January 1987

"Feasibility of a Mapping from the Ada Formal Definition to the NYU SETL Interpreter for ADa", January 1987

"On the Feasibility of the execution of the ADa Formal Definition", January 1987

T. Gallo et al.

"Ada FD Tool Set: Architecture and Design, June 1986

"Ada FD Tool Set: Architecture and Preliminary Design", September 1986

"Ada FD Tool Set· Design", January 1987

"Requirements for a Portable AdaFD Tool Set", January 1986

A Giovini: "Towards a Formal Specification of a Static Checker for the Ada Formal Definition", September 1986

A Giovini et al.

"The User Manual of the Document Metalanguage", September 1986

"Tasking - Using a Direct Semantics Sytle", July 1985

S.Gnesi et al.

"On Selecting the Specification Language(s): Guidelines", August 1985

"Tentative Specification Language: Guidelines", July 1985

K.W. Hansen et al.

"Example 'Difficult' Subset of Ada", July 1985

"The Draft Formal Definition of Ada, Exploitation Plan",

E.W. Karlsen: "Correlation ANSI Ada - Ada FD", March 1987

G. Reggio: "A Direct Semantics Sytle for D-SMoLCS", January 1985

"Tiral Metalanguage for Algebraic and Applicative Parts", September 1985

G. Reggio et al. "The User Manual of the Metalanguage", December 1986

B. Scognamiglio: "Ada FD Toolset: Primer", April 1987

B. Scognamiglio: "Ada FD Toolset: Installation & User Manual", April 1987

J. Storbank Pedersen: "Static Semantics Example Ada Subset", August 1985

3.7 Published Papers Related to The Project

The following are by Astesiano et al.:

"On the Parameterized Algebraic Specification of Concurrent Systems", in Proc. CAAP '85 - TAPSOFT Conference, Lecture Notes in Computer Science, Vol. 185 pp. 342 - 358, Springer Verlag, 1985.

"Formal Specification of a Concurrent Architecture in a Real Project", in Proc. of ICS '85, (ACM International Computing Symposium), pp.185 - 195, North Holland, 1985.

"A Syntax-directed Approach to the Semantics of Concurrent Languages", in Proc. 10th IFIP World congress (H.J. Kugler ed.), pp.571-576, North HOlland, 1986.

"Relational Specifications and Observational Semantics", in Proc. of MFCS' 86, Lecture Notes in Computer Science, Vol. 233, pp. 209 - 217, Springer Verlag, 1986.

"The Ada Challenge for New Formal Semantic Techniques", in Ada: Managing the Transition, in Proc. of the Ada-Europe Cambridge University Press, 1986.

"The SMoLCS Approach to the Formal Semantics of Programming Languages - A Tutorial Introduction", to appear in Proc. of CRAI Spring

International Conference: Innovative Software Factories and Ada, Capri, 1986, Lecture Notes in Computer Science, Springer Verlag, 1987.

"Comparing Direct and Continuation Styles for Concurrent Languages" in Proc. STACS '87, Lecture Notes in Computer Science, Vol. 247, pp. 311 - 322, Springer Verlag, 1987.

"SMoLCS-Driven Concurrent Calculi" in Proc. TAPSOFT '87, Lecture Notes in Computer Science, Vol. 249, pp. 169 - 201, Springer Verlag, 1987.

"Semantics of Concurrent Languages in the SMoLCS Approach", to appear in IBM Journal of Research and Development, 1987.

"Definition Semantique Formelle du Langage Ada, une Application de la Methodologie SMoLCS", in Logiciels: Mergence des Normes, Enjeux Vol. 75, pp. 37 - 39, December 1986.

"An Outline of the SMoLCS Methodology", to appear in Proceedings of Advanced School on Mathematical Models for Parallelism, Roma 1986, Lecture Notes in Computer Science, Springer Verlag, 1987.

"An Introduction to the Draft Formal Definition of Ada", in Proceedings of the 3d IDA Workshop on the Formal Specification and Verification of Ada, Research Triangle Park, May 1986.

K.W. Hansen: "Structuring the Formal Definition of Ada", in Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station, Houston, June 1986.

J. Storbank Pedersen: "VDM in Three Generations of Ada Formal Descriptions", in VDM'87, VDM - A Formal Method at Work, Lecture Notes in Computer Science, Vol. 252, Springer Verlag, March 1987.

Chapter 4

Critical Evaluation from a U.S. Perspective

4.1 What was Accomplished?

Several benefits accrued from the limited U.S. participation in the Advisory Group review of the AdaFD. The effects of this participation are discussed below.

4.1.1 Coordination with U.S. Ada Activities

In response to a strong suggestion by the U.S. members of the Advisory Group, the Ada FD team coordinated their effort with the work of the Ada Language Maintenance Committee (LMC) which meets periodically to consider ambiguous aspects of the Language Reference Manual. The Ada FD team modified their Formal Definition in the light of the various rulings of the LMC. For example, in an earlier version of the FD, there was an attempt to model time in order to formally define the Ada "delay" statement. When confronted with the problem of informally defining the meaning of "delay," the LMC has refused to be precise. Instead, they have said words to the effect that a compiler correctly compiles a "delay n" if the machine does delay a time "reasonably" close to n. Of course, the LMC makes no attempt to define "reasonably." The FD could lead the LMC here by providing a mathematical model of "delay," but this was felt not to be an appropriate area in which to display the power of formal methods since real-time is of notorious difficulty from the mathematical point of view. Instead, the FD was revised to omit its modeling of real-time. The Advisory Group felt that this was too extreme and that the Formal Definition should support the inclusion of tentative or alternative definitional clauses which are not being presented as definitive, but only as provisional. Alternative modelings of real-time could then be provided on such a provisional basis.

The FD team followed very closely the various LMC positions on tasking. Such issues include: termination of unactivated tasks; effect of priorities on selective waits; rendezvous between conditional entry call and select with else part; when termination takes place as a modification of the attribute; queuing conditional timed entry calls; exceptions in an abnormally completed task; atomicity of actions (e.g., activating process, execution of abort statement); synchronization points including task attributes and renaming of entries.

On the whole, it was felt that the strong recommendation to network with the LMC and its subsequent facilitation was one of the main positive contributions of the Advisory Group to the FD effort.

In addition, U.S. participation in the Advisory Group led to members of the Ada FD team being invited to present their work at U.S. forums. They contributed to the various IDA Workshops on the Formal Specification and Verification of Ada and at several SIGAda meetings. At an IDA Workshop, a whole day was devoted to the FD and at SIGAda several evening meetings of the Formal Methods Committee in addition to general 1-hour talks to the attendees at large.

Such visibility brought them into contact with diverse subgroups of the Ada world such as people with real-time concerns. In Europe, it appears, separation between theoretical, academic investigations and industrial practitioners is stronger than in the U.S. The lack of an entrepreneurial spirit sentences the mathematically inclined to a lifelong university career. The exposure to these non-theoretical concerns was very beneficial.

4.1.2 Concern with the Complexity of the FD

The issue of the complexity of the underlying methodology was repeatedly raised in the Advisory Group primarily by myself. As a logician specializing in these areas, I was fully conversant with all the material which went into the FD; the other U.S. participants were less so. Hence, they were reluctant to criticize what they didn't fully understand and were more willing to accept the FD's team argument that the complexity was necessary. As we have seen, a thorough mastering of the FD requires an understanding of denotational semantics, algebraic semantics, and the Milner synchronization tree approach to concurrency. Indeed, I raised the need for a basic theorem to show that SMoLCS really combines these disciplines correctly. Joe Stoy, the Chairperson of the Advisory Group, agreed that such a theorem is required and added this request to the list of action items.

The purpose of the meta-language, which sits on top of SMoLCS, is to present the definition in a comprehensible manner. The meaning of the meta-language constructs are given in terms of SMoLCS. The coherence theorem requested is needed to show that the underlying SMoLCS meaning of a meta-language construct agrees with the reader's intuitive understanding of it. This is necessary since most users will interact with the FD only at the meta-language level.

As a response to the complexity criticisms, several expository papers were produced by Astesiano and his coworkers.

In fairness, it should be mentioned that the complexity problem arose from the fact that the basic theoretical work of developing a formal speci-

fication language adequate for modeling Ada and the actual modeling were performed concurrently over a short period. As a result, the underlying methodology appears to be pieced together with spit and chewing gum. Put another way, the present European Ada FD can be considered as a prototype proving feasibility.

4.1.3 Criticism of the Approach to Tool Sets

A good portion of the presentations to the Advisory Group concerned various tool sets which are being proposed to manipulate the FD. While the need for such tools is manifest, I was dismayed that none of the proposals required the use of Ada in tool construction. From a U.S. perspective this is weird, if understandable. The European position is that they have no Ada world (programmers, compilers, environments, etc.) comparable to the U.S. and, therefore, building a significant Ada program (i.e., the FD tools) is beyond their capability. Such a response lends credence to the criticism that the FD is largely a bookish, academic exercise with little ties to real software engineering. Alternatively, one can view the lack of technology as a blessing; it has forced a higher level of mathematics than usual. The Genoa group, for example, had no direct access to an Ada compiler. All tools being proposed were adaptations of exisiting tools such as Gandalf from CMU. Furthermore, the people who presented the tool set plans appeared much less competent than comparable U.S. tool builders with whom we were familiar. In contrast to their strong theoretical bent, there is obviously a lack of software experimentation in Europe.

4.1.4 Improvement of the Literary Style

Further points were raised about literary style. All of the available documentation was produced by non-native speakers of English. In quite a few places, this acts as a major obstacle for the reader; this is particularly so for Astesiano's theoretical portions. While this Advisory Group comment is certainly valid, it is difficult to see how a European effort could address it. What is needed is a major introduction to the FD written in colloquial American-English. Which brings us to the next portion of our report.

4.2 What is to be Done?

The Ada FD represents the largest formal specification of which I am aware. How is it to be debugged, how is to be used, how is it to be upgraded? Various recommendations follow.

4.2.1 A Gentle Introduction

The purpose, approach, framework, achievements and neglects of the FD could be presented in an expository manner similar to a textbook or a book in the Lecture Notes in Computer Science series. None of the existing documentation serves this purpose. When asked to produce something more expository, the FD team produces dense, difficult to read articles which compound the problem. The main difficulty appears to be the team's lack of native English speakers together with the advanced nature of the theoretical computer science involved in the FD. Before the FD can be debugged, such a work should be completed to guide the Ada critics. To be really complete, the Introduction should include within it all the material necessary to understand the FD. This places it at textbook proportions.

4.2.2 Tools

To navigate in the FD, tools are necessary. The current European efforts to produce such tools (I am not really sure of the status of these efforts) will not suffice for reasons mentioned earlier. Indeed, it would seem to me that tools written in Ada and conforming to CAIS are a necessity for U.S. usage. Because of export issues involving Ada environments, it might not be best to have a European group do this effort (in particular Europe is developing its own tool interface standard). A useful component of such tools is the ability to generate English text from meta-language formulas to aid reader comprehension. Simulation and theorem proving components would also be useful to help the user explore consequences of various FD clauses.

4.2.3 Debugging

Since humans are error prone, it is customary to separate the quality control phase of engineering efforts from the main development phase. The FD should not be debugged by the authors of the report. The Introduction and Tools mentioned in the previous recommendations are primarily for the debuggers. Unlike a compiler, which can run test suites, the debugging

of the FD is largely a manual exercise at the current stage of technology. Whether the effort required for such an undertaking is commensurate with the expected benefits is a policy decision. It is my opinion that it is.

4.2.4 A New Effort

It is not out of the question to view the European effort as a rapid prototype and to undertake a major revision. Such an effort could be pursued by a joint U.S./European team using some of the original key players. It would start by considering which parts of the underlying methodology turned out to be really essential for the resulting definition. In this way, the underlying methodology could be simplified and fine tuned to the problem at hand. It is expected that large parts of the actual FD in the meta-language could remain intact. Hence, the previously mentioned efforts whose goal was to facilitate debugging of the definition are not negated by such a reshaping of the foundations.

If such an undertaking were to succeed, the issue of tools would have to be addressed from the start. Feasibility of navigating and executing the resulting definition should be a major driver. David Luckham and myself have had long discussions on how this could be approached. However, details regarding these approaches are beyond the scope of this report.

References

[ANS83] ANSI. *The Programming Language Ada Reference Manual. Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1983.

[BHR84] Brookes, Hoare, and Roscoe. Theory of communicating sequential processes. *Journal of the ACM*, 31(3), 1984.

[Bjo80] D. Bjorner. *Towards a Formal Description of Ada. Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1980.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer-Verlag, Berlin, 1985.

[Jon80] N.D. Jones. *Semantics-Directed Compiler Generation. Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1980.

[Les87] P. Lescanne. *Rewriting Techniques and Applications. Proceedings 1987.* Volume 256 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1987.

[Mil80] Robin Milner. *A Calculus of Communicating Systems.* Volume 92 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1980.

[Mil85] Robin Milner. Lectures on a calculus for communicating systems. In Manfred Broy, editor, *Control Flow and Data Flow: Concepts of Distributed Programming*, pages 205–228, Springer-Verlag, 1985.

[RT84] Thomas Reps and Tim Teitelbaum. The synthesizer generator. In *Proceedings of ACM SIGSOFT/SIGPLAN Software Enigineering Symposium on Practical Software Development Environments*, pages 42–48, Association for Computing Machinery, SIGPLAN, Balitimore, MD, April 1984.

[SDS86] J. T. Schwartz, R. B. Dewar, and E. Schonberg. *Programming with sets; an introduction to SETL.* Springer-Verlag, Berlin, 1986.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Massachusetts, 1977.

Distribution List for IDA Memorandum Report M-389

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
Dr. John Solomond Director Ada Joint Program Office Room 3D139, The Pentagon Washington, D.C. 20301	2
Mr. John Faust Rome Air Development Center RADC/COTC Griffiss AFB, NY 13441	2
Other	
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
IIT Research Institute 4550 Forbes Blvd., Suite 300 Lanham, MD 20706	1
Dr. Richard Platek Odyssey Research Associates 1283 Trumansburg Rd. Ithaca, NY 14859-1313	2
Mr. Karl H. Shingler Department of the Air Force Software Engineering Institute Joint Program Office (ESD) Carnegie Mellon University Pittsburgh, PA 15213-3890	1
IDA	
General W.Y. Smith, HQ	1
Ms. Ruth L. Greenstein, HQ	1
Mr. Philip L. Major, HQ	1
Dr. Robert E. Roberts, HQ	1
Ms. Anne Douville, CSED	1
Dr. Richard J. Ivanetich, CSED	1
Mr. Terry Mayfield, CSED	1

NAME AND ADDRESS	NUMBER OF COPIES
Ms. Katydean Price, CSED	2
Dr. Richard Wexelblat, CSED	1
IDA Control & Distribution Vault	3